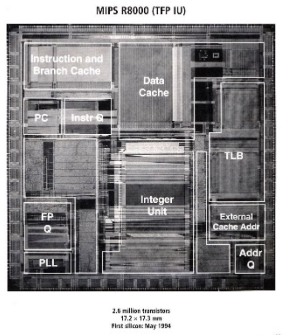
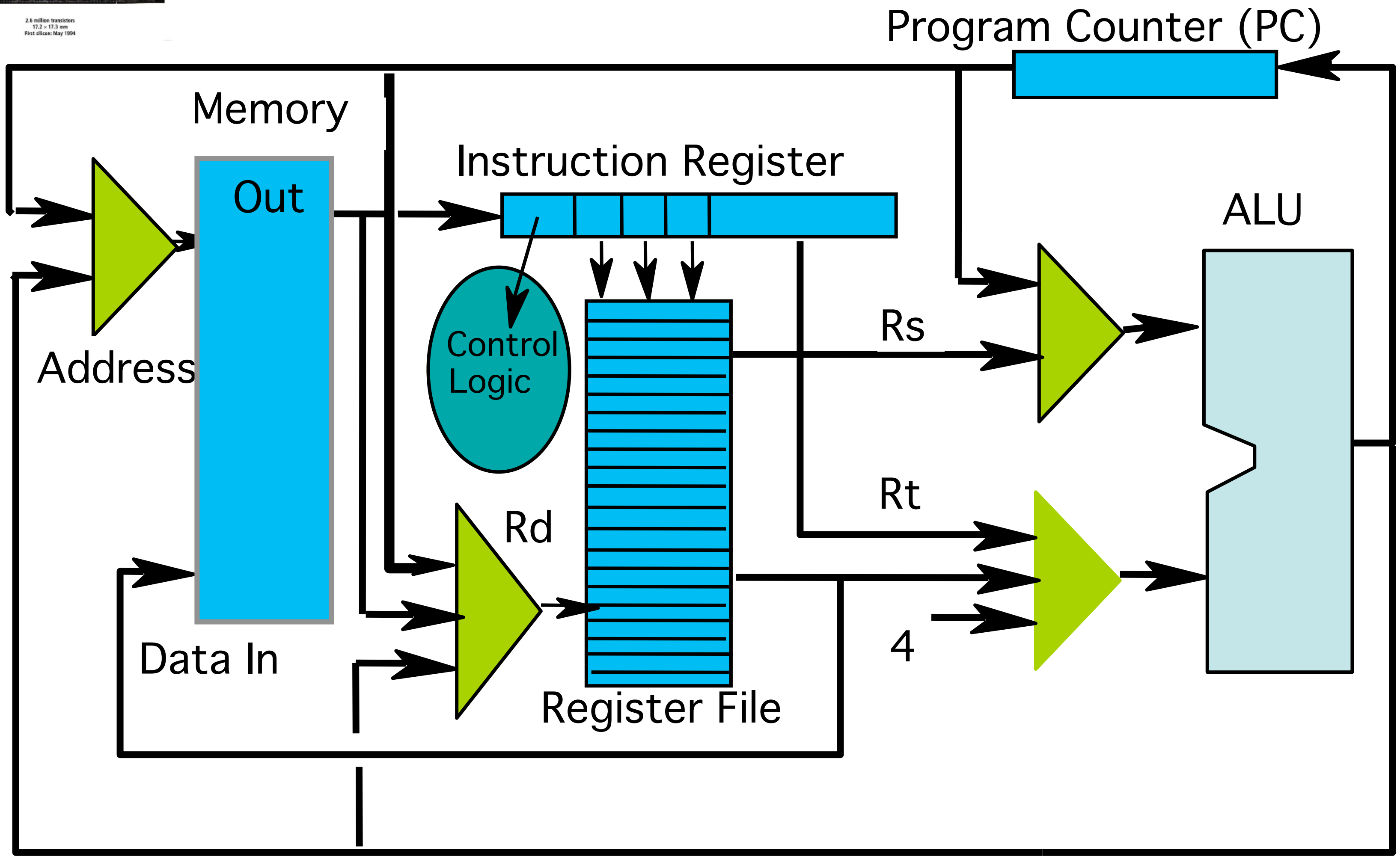
MIPS Pipeline Details

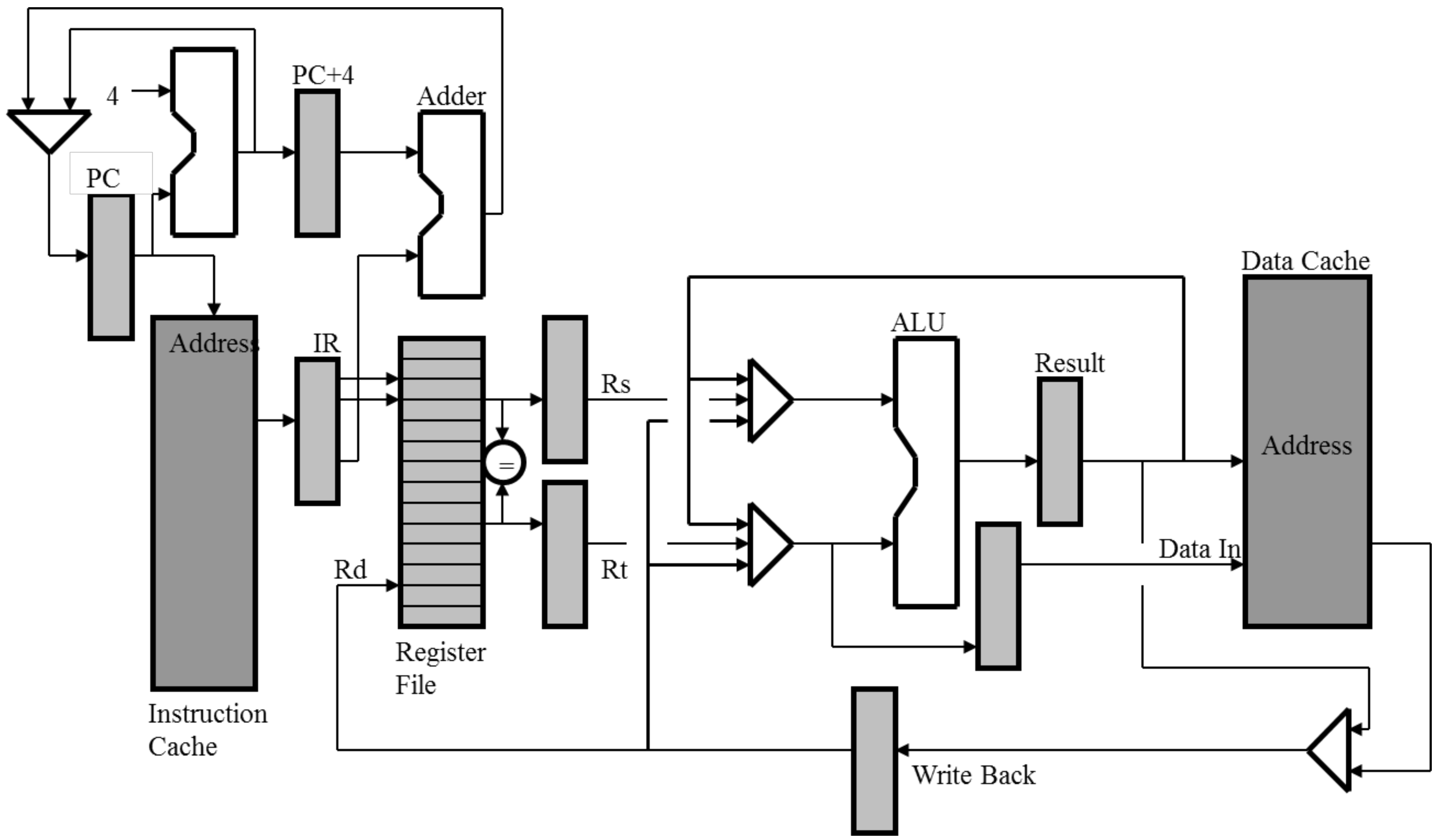
Dr. Edward Nava
ejnava@unm.edu



MIPS Data Path Diagram



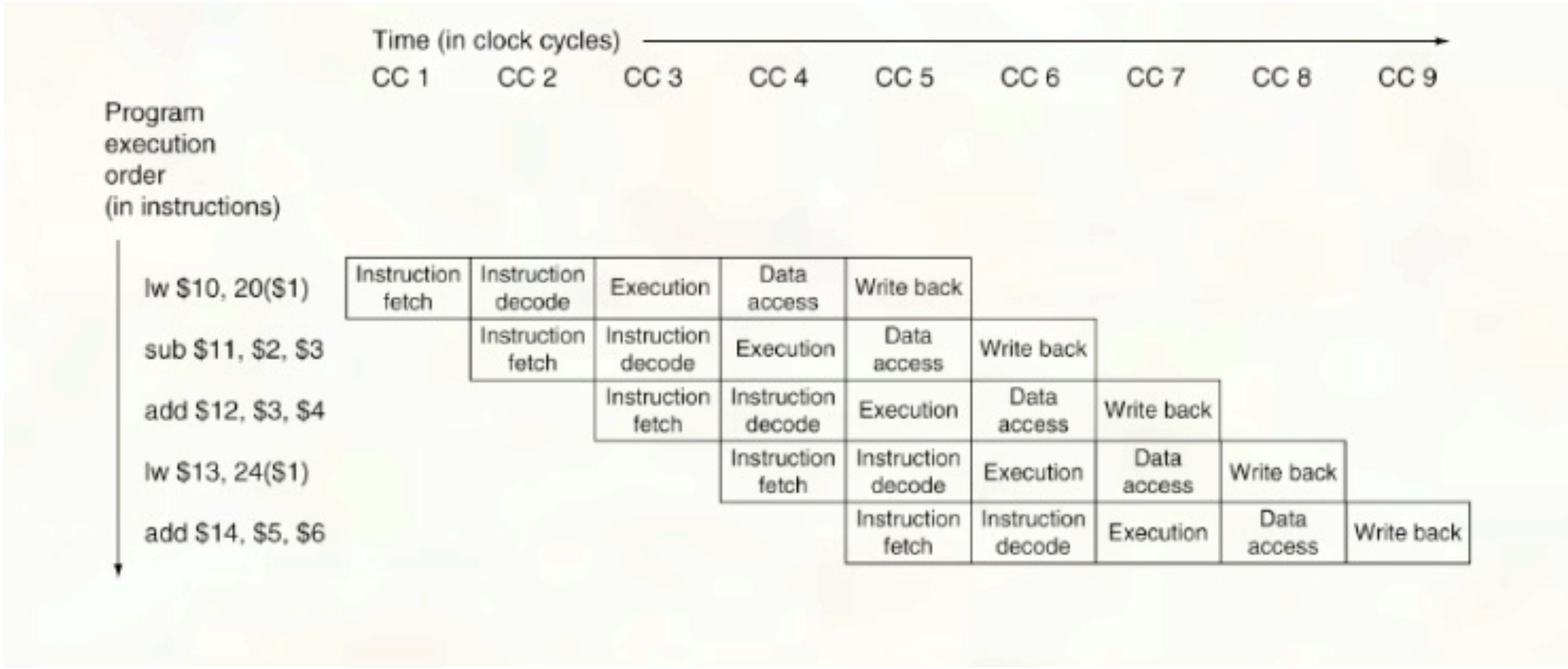
Simplified MIPS Pipeline



Assembly Line Processing Concept

Modern Assembly Line

MIPS Pipelined Cycle Execution



Time (in clock cycles) →

Program execution order (in instructions) ↓

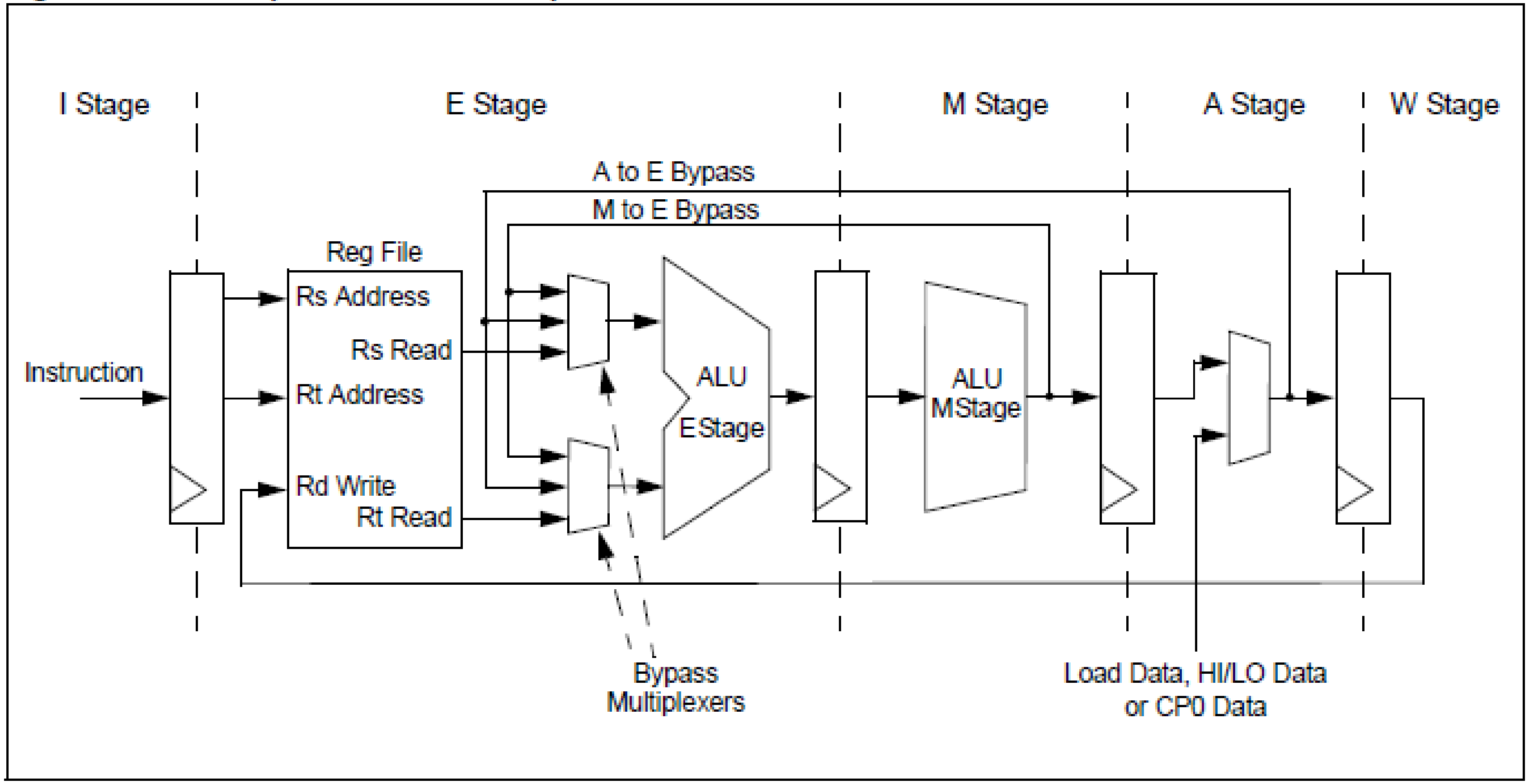
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

MIPS Five-Stage Pipeline

1. Instruction Fetch Stage - Fetch the instruction from cache memory and load it into the Instruction Register (IR). Increment the Program Counter (PC) by four.
2. Operand Fetch Stage - Fetch values Rs and Rt from the Register File. If this is a branch instruction and the branch condition is met, then load the PC with the branch target address.
3. Execute Stage - Perform an arithmetic or logic function in the ALU and load the Result register. This is the stage where an addition is performed to calculate the effective address for a load or store instruction.
4. Memory Access Stage - If the instruction is a load, a read from the data cache occurs. If the instruction is a store, write to the data to cache occurs. Otherwise, pass the data in the Result register on to the Write Back register.
5. Write Back Stage - Store the value in the Write Back register to the Register File. Notice that when the first instruction into the pipeline is storing results back to the Register File the fourth instruction into the pipeline is simultaneously reading from the register file.

MIPS CPU Pipeline

Figure 2-3: Simplified PIC32 CPU Pipeline



The results of using instruction pipelining in the PIC32 core is a fast, single-cycle instruction execution environment.

Data Hazards

- **Data hazards** occur when the values needed as operands for an instruction have not yet been written back into the register file.
- This occurs when the result of one instruction is used as an operand in the next instruction(s)
- One hardware solution for this issue is the use of forwarding paths in the machine's data paths.
- The hardware forwarding solution will not work in the cases where load from memory instructions are used. - This requires rearranging assembly language instructions or the insertion of nops to ensure that data are available for instructions that immediately follow a load instruction.

Control Hazards

- Associated with every branch or jump, we have a **control hazard**.
- A branch or jump will take effect after the instruction following the branch or jump is in the pipeline.
- This can sometimes be dealt with by rearranging the code to place an instruction in the **delay slot** following the branch
- If rearranging is not feasible, then a nop is used after the branch or jump.

Using SPIM to simulate pipelined operation

- To examine the effects of delayed branches and delayed load, SPIM can be configured to do so.
- Under the **settings** options, select **Delayed Branches** and **Delayed Loads.**

```
   prompt: .asciiz        "\nPlease Input a value:"
    result:      .asciiz        "  The sum Integers="
   bye:          .asciiz        "\n  **** Adios Amigo - "
              .text
main:
        li              $v0, 4              #
              la $a0, prompt              #
                syscall            #
        li              $v0, 5              #
                syscall            #
          blez             $v0,  End        #
        li              $t0, 0              #
loop:
           add               $t0, $t0, $v0     #
           addi              $v0, $v0, -1      #
          bnez            $v0,  loop           #
        li              $v0, 4             #
                  la $a0, result #
                syscall            #
        li              $v0, 1              #
          move            $a0,  $t0            #
                syscall            #
          b               main             #
End:           li          $v0, 4                 #
       la               $a0, bye            #
                syscall            #
```

```
                  .text # For a Pipelined Implementation
main:
        li              $v0, 4                  #
              la $a0, prompt              #
                syscall            #
        li              $v0, 5                  #
                syscall            #
          blez             $v0,  End             #
          move            $t0, $v0              #***
loop:
            addi              $v0, $v0, -1       #
          bnez            $v0,  loop             #
              add                   $t0, $t0, $v0
                                  #***
        li              $v0, 4                  #
              la                $a0, result


                syscall            #
        li              $v0, 1                  #
          move            $a0,  $t0              #
                syscall            #
        b               main                  #
   End:        li            $v0, 4                   #
        la                $a0, bye               #
                            syscall
```

Section 10.8

```
#######################################
# $a0: Pointer to Array
# $a1: Number of elements
#######################################
Sum:    li      $v0, 0              #
        li      $v1, 0              #
Loop:
        blez    $a1, Return         #
        addi    $a1, $a1, -1        #
        lw      $t0, 0($a0)         #
        addi    $a0, $a0, 4         #
        bltz    $t0, Negative       #
        add     $v0, $v0, $t0       #
        b       Loop                #
Negative:
        add     $v1, $v1, $t0       #
        b       Loop                #
Return:
        jr      $ra                 #
```

```
######### Modified for Pipelined I#########
# $a0: Pointer to Array
# $a1: Number of elements
#######################################
Sum:    li      $v0, 0
        li      $v1, 0
Loop:
        blez    $a1, Return         #
        lw      $t0, 0($a0)         #
        addi    $a0, $a0, 4         #
        bltz    $t0, Negative       #
        addi    $a1, $a1, -1        # ***
        b       Loop                #
        add     $v0, $v0, $t0       # ***
Negative:
        b       Loop                #
        add     $v1, $v1, $t0       # ***
Return:
        jr      $ra                 #
        nop                         # ***
```

```
•           .text
•     MinMax:
•           lw          $v0, 0($a0)  #
•           addiu       $a0, $a0, 4   #
•           move        $v1, $v0                   #
•           addi        $a1, $a1, -1 #
•           blez        $a1, ret                   #
•     loop:
•           lw          $t0, 0($a0)   #
•           addi        $a0, $a0, 4   #
•           bge         $t0, $v0, next             #
•           move        $v0, $t0                   #
•           b           chk                        #
•     next:                                        #
•           ble         $t0, $v1, chk              #
•           move        $v1, $t0                   #
•     chk:                                         #
•           addi        $a1, $a1, -1 #
•           bnez        $a1, loop                  #
•     ret:
•           jr          $ra                        #
```

```
          .text  # Pipelined Implementation
MinMax:
          lw          $v0, 0($a0)  #
          addiu       $a0, $a0, 4  #
          addi        $a1, $a1, -1 #
          blez        $a1, ret                 #
          move        $v1, $v0                 #***
loop:
          lw          $t0, 0($a0)  #
          addi        $a0, $a0, 4  #
          bge         $t0, $v0, next           #
          nop                                  #***
          b           chk                      #
          move        $v0, $t0                 #***
next:                                          #
          ble         $t0, $v1, chk            #
          nop                                  #<<<<<
          move        $v1, $t0                 #
chk:                                           #
          addi        $a1, $a1, -1 #
          bnez        $a1, loop                #
          nop                                  #***
ret:
          jr          $ra                      #
          nop                                  #***
```

The True Branch Instructions

Branch if Equal:	beq	Rs, Rt, Label
Branch if Greater Than or Equal to Zero:	bgez	Rs, Label
Branch if Greater Than or Equal to Zero & Link:	bgezal	Rs, Label
Branch if Greater Than Zero:	bgtz	Rs, Label
Branch if Less Than or Equal to Zero:	blez	Rs, Label
Branch if Less Than Zero and Link:	bltzal	Rs, Label
Branch if Less Than Zero:	bltz	Rs, Label
Branch if Not Equal:	bne	Rs, Rt, Label